DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming - Zombi defense

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
Rey Maximiliano	37/13	rey.maximiliano@gmail.com
Tirabasso Ignacio	718/12	ignacio.tirabasso@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Ejercicio 1

a) Tabla de descriptores de la GDT.

Al momento de armar la GDT, acorde a lo dispuesto en el enunciado, dejamos las primeras siete entradas de la tabla de descriptores libres. Luego, a partir de la posición 8 dejamos los descriptores de segmento pedidos: código y datos nivel 0 y código y datos nivel 3.

Armamos los cuatro segmentos de la GDT, llamándolos:

```
[GDT_IDX_DATA_0] = (gdt_entry) ;
[GDT_IDX_DATA_3] = (gdt_entry) ;
[GDT_IDX_CODE_0] = (gdt_entry) ;
[GDT_IDX_CODE_3] = (gdt_entry) ;
```

A los cuatro les seteamos el mismo *límite*: **0x26EFF** y la misma *base* en **0**, de este modo los cuatro descriptores de segmento direccionan a los primeros *623MB* de memoria. El *segment type* varía depende el segmento: CODE_0: **0x0A** (code execute/read), CODE_3: **0x0F** (code execute/read, conforming, accesed), DATA_0 y DATA_3: **0x02** (data read/write). El *Descriptor type* va en todos para system, por lo tanto es **0**. El *Descriptor privilege level* coincide con el nombre del descriptor (**0** para CODE_0 y DATA_0; **3** para CODE_3 y DATA_3). El bit de *Present* va para todos en **1** y los bit de *Available for use by system software* y *l* van para todos en **0**. El bit de *Default operation size* va para todos en **1** porque es un código de 32bits. El bit de *Granularity* va para todos en **1**.

b) Pasaje a modo protegido y seteo de la pila del kernel.

Para pasar a modo protegido los pasos que debemos llevar a cabo son:

- ▷ Completar la GDT (resuelto en el inciso A).
- ▷ Deshabilitar interrupciones (para ello se ejecuta la instrucción *cli*).
- ▷ Habilitar A20 (en nuestro caso se resuelve haciendo *call habilitar_A20*).
- ▷ Cargar el registro GDTR con la dirección base de la GDT (lo hacemos con la instrucción *lgdt [GDT_DESC]* la etiqueta GDT_DESC apunta al descriptor de la GDT en el código).
- ▷ Una vez hecho esto, estamos en condiciones de setear el bit PE del registro CR0 (debemos hacer: *mov eax,cr0 ; or eax,1 ; mov cr0,eax*).
- ▷ Lo siguiente a realizar es el *jump far* a la siguiente instrucción (utilizamos el selector de segmento *0x50* y de offset la etiqueta *modo_protegido* por que el selector es 0x50??).
- ▷ Una vez ya en modo protegido, nos encontramos trabajando en 32 bits y ahora es cuando cargamos los registros de segmento (a los registros *es, ds, ss y gs* les asignamos el valor de *0x40* y al registro *fs* le asignamos el valor *0x60* aca habría que poner el porque de estos valores??).

Para setear la pila del kernel en la dirección 0x27000 debemos llevar a cabo la siguiente instrucción:

```
mov ebp, 0x27000
mov esp, 0x27000
```

c) Segmento adicional que describe el área de la pantalla en memoria que puede ser utilizado sólo por el kernel.

HELP! aca no se que ponerrrrrrrr

```
; Cambiar modo de video a 80 X 50
mov ax, 0003h
int 10h ; set mode 03h
xor bx, bx
mov ax, 1112h
int 10h ; load 8x8 font
```

d) Rutina que se encarga de limpiar la pantalla y pintar el área del mapa

En este punto debemos establecer un fondo de color verde, junto con las dos barras laterales para cada uno de los jugadores (una roja y otra azul). Para esto, debemos contar con una función que limpie la pantalla en un primer momento: *clear_screen*.

La función ***clear_screen***, implementada en lenguaje C, se va a encargar de:

- Guardar en una variable local: *size* el tamaño de la pantalla (VIDEO_COLS * VIDEO_FILS).
- Luego va a hacer un while desde *0* hasta *size* que, empezando por la dirección donde está almacenada la memoria de video, vaya guardando el caracter que es todo negro.

De este modo, logramos hacer que toda el área de la pantalla quede “pintada” de negro.

En segunda instancia, armamos la función *print_map*. La función ***print_map***, también implementada en lenguaje C, con el fin de pintar el área del mapa con los colores deseados, posee el siguiente comportamiento:

- En un primer momento, llama a la función *clear_screen*.
- Se arman cuatro variables locales: blue, red, green y black. Cada una de ellas es un caracter completamente de su color.
- Mediante dos *fors* anidados recorremos toda el área del mapa, y dependiendo de la posición en la que se encuentre es el color que le va a ser asignado (rojo a las primeras dos columnas de la izquierda, azul a las últimas dos, negro a las últimas cinco filas y el resto en verde).

Luego este código fue complejizado para que pueda imprimir los puntos de los jugadores y sus respectivos puntajes por zombie tal cual debe ser cuando comienza el juego.

2. Ejercicio 2

a) Entradas en la IDT.

Inicializamos las entradas 0-19 en la IDT (todas con privilegio 0). Creamos un arreglo de strings, que contenga los nombres de cada excepción para luego mostrarlo en pantalla como se ve a continuación:

```
static char *exceptions[] = {
    "Division by zero",
    "Debugger",
    "NMI",
    "Breakpoint",
    "Overflow",
    "Bounds",
    "Invalid Opcode",
    "Coprocessor not available",
    "Double fault",
    "Coprocessor Segment Overrun (386 or earlier only)",
    "Invalid Task State Segment",
    "Segment not present",
    "Stack Fault",
    "General protection fault",
    "Page fault",
    "reserved",
    "Math Fault",
    "Alignment Check",
    "Machine Check",
    "SIMD Floating-Point Exception"
};
```

El código de cada rutina de excepción lo que va a hacer es:

Guardar en memoria el estado de los registros de la tss actual para luego pushearlos y llamar a la función *print_exception*. Esta función va a estar encargada de imprimir el nombre del problema que se produjo, conjunto a todos estos registros que nos habíamos almacenado.

b) Prueba de lo armado en el punto A.

Para que el procesador utilice la IDT creada anteriormente, generamos la excepción “División por cero” para probarla:

```
    ; Inicializar la IDT
    call idt_inicializar
    ; Cargar IDT
    lidt [IDT_DESC]

    ; test para que salte la divide by 0 exception (0)
    mov edx,0
    mov ecx,0
    mov eax,3
    div ecx
```

Efectivamente, anduvo acorde a lo esperado.

3. Ejercicio 3

a) Implementación completa de print_map.

En el ejercicio 1 ya habíamos logrado armar **clear_screen** y **print_map**, esta última contaba con una funcionalidad acotada. Sólo armaba cuatro bloques de colores. Ahora lo que vamos a hacer es extender la función **print_map** para que escriba los puntos de los jugadores y marque el estado de los zombies, tal como aparece en la *figura 9* del enunciado.

La función print_map la vamos a extender sumándole las siguientes instrucciones:

- Debemos armar los *cuadrados de puntaje* para el jugador rojo y para el jugador azul. Nos movemos entre las últimas cinco filas, desde la columna 35 hasta la 39 se pintan con el caracter completamente rojo y desde la columna 40 hasta la 44 se pintan con el caracter completamente azul.
- Escribimos en el centro de cada cuadrado el puntaje inicial: 0. Para ello creamos dos caracteres que sean 0 y cada uno con los atributos necesarios para que sean: fondo rojo, letra blanca y fondo azul, letra blanca.
- Luego escribimos la cantidad de zombies restantes, para lo cual se utilizan dos caracteres idénticos a los mencionados arriba. Estos puntajes se ubican cada uno a un costado de su cuadrado respectivo.
- Por último, resta escribir el estado de los zombies. Generamos un char* que sean todos los números de zombies y dándole formato de fondo negro, caracter blanco los copiamos dos veces: uno para el jugador azul y otro para el jugador rojo. Como todos los zombies se encuentran disponibles, generamos otro char* que sean diez 'x' y ubicamos en su posición correspondiente uno que posea los atributos de fondo negro, caracter azul y otro de fondo negro, caracter rojo.

Con el objetivo de hacer que lo descripto anteriormente sea una tarea más simple contamos con las funciones: *print_string* y *get_format*.

La función **print_string** recibe una posición en el mapa (x,y), un char* con el texto que queremos imprimir en pantalla y un short con los atributos deseados. PONER ALGO DE AVOID PRINT BUG. Dentro de un *for* que recorre horizontalmente la posición en memoria a partir de la posición (x,y), se va avanzando el char* y en cada iteración se le asigna a esa posición de memoria el char actual con los atributos pasados por parámetro.

La función **get_format** es una función simple la cual recibe como parámetro los atributos deseados y los devuelve con el formato de un sólo char, que es el que debemos utilizar.

```
unsigned char getFormat(unsigned char fore_color, char fore_bright, unsigned char back_color,
                        char blink) {
    return fore_color | fore_bright | back_color | blink;
}
```

b) Rutinas encargadas de inicializar el directorio y tablas de páginas para el kernel.

Para poder mapear las direcciones 0x00000000 a 0x003FFFFF es necesaria una sola entrada en el Page Directory. En un primer momento, se crea un puntero a Page Directory en la dirección 0x27000. Se limpian las 1024 entradas del Page Directory poniendo todos sus bits en 0.

Luego, a la primera entrada del Page Directory (índice: 0) se le asigna como base 0x28. Se le asigna permiso de escritura y lectura con su bit de presente seteado, como es de Kernel es de nivel 0.

Para que todo esto funcione y el primer índice del Page Directory apunte a un Page Table válido, se debe crear un Page Table en la posición 0x28000 en memoria. En esta misma se completan todos los

índices (ya que es lo que ocupa el rango pedido para el Kernel) direccionándolos desde la posición 0x0 en memoria, aumentando en uno acorde aumenta el índice. Todos tienen permiso de escritura con el bit de presente seteado, como es de Kernel es de nivel 0.

c) Código necesario para activar paginación.

Como ya contamos con un directorio de páginas y una tabla de páginas estamos en condiciones de activar paginación.

Como para esto debemos poner en cr3 la base del directorio de páginas y limpiar los bits PCD y PWT del mismo, basta con asignarle a cr3 el valor de 0x27000. De este modo la base queda en 0x27 y los bits limpios. Es correcto esto?

Por último resta setear el bit PG del cr0, lo hacemos mediante un or.

4. Ejercicio 4

a) Administración de la memoria en el área libre.

Como conocemos las limitaciones de memoria que se poseen y acorde el contexto de uso, nunca vamos a pedir más páginas de las existentes porque antes terminaría el juego. Por este motivo, implementamos un contador llamado *páginas* que denota la cantidad de páginas utilizadas de modo que administra la memoria en el área libre.

Luego contamos con las funciones *get_page_directory* y *get_page_table* las cuales nos brindan un nuevo page directory o una nueva page table correspondientemente. Ambas se basan en esta variable que poseemos.

La función **get_page_directory** toma la variable global *páginas* y la multiplica por 4096 (el tamaño) y se lo suma a la dirección PAGES (que es 0x100000), siendo el número obtenido el puntero al nuevo Page Directory. Se inicializa en 0 cada índice de los 1024 que son y luego a la primer posición se le da como base 0x28, se le da permiso de lectura/escritura y se setea el bit de presente (por que hacemos esto con la primer posicion?). Por último se incrementa en uno el contador de páginas.

La función **get_page_table** tiene un comportamiento similar: adquiere el puntero a la nueva página de la misma manera, inicializa en cero sus 1024 entradas y por último incrementa en uno el contador de *páginas*.

Es decir, a partir de la dirección 0x100000 se arman todos los Page Directories y Page Tables.

La función **inicializar_mmu** tiene un comportamiento simple: llama a la función *mmu_inicializar_dir_kernel()* y luego pone en cero el contador *páginas*.

b) mmu_inicializar_dir_zombi

La rutina *mmu_inicializar_dir_zombi* se encarga de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 6. Copia el código de la tarea a su área asignada, es decir la posición indicada por el jugador dentro del mapa y mapea dichas páginas a partir de la dirección virtual 0x08000000(128MB).

La función **mmu_inicializar_dir_zombie** recibe como parámetro el jugador (si es A o B), la clase y su posición en el eje de las *y*.

- ▷ Va a pedir un nuevo Page Directory mediante la función explicada anteriormente: *get_page_directory*.
- ▷ El valor de x va a ser ANCHO_MAPA-2 si el jugador es B, 1 si el jugador es A.
- ▷ Luego de eso, se mapean las nueve páginas correspondientes utilizando mmu_mapear_pagina y como función auxiliar get_physical_address. Esta última nos permite pasarle un (x,y) en el mapa y nos da la dirección física del mismo.
- ▷ Consecuentemente, copiamos el código en la página central.
- ▷ Devolvemos el pd creado.

c) Mapear y Des-Mapear página.

La rutina *mmu_mapear_página* permite armar toda la estructura necesaria para que, dada una dirección virtual, un puntero a Page Directory, una dirección física y los atributos deseados (lectura/escritura

y nivel); se mapee la dirección virtual a la física.

El comportamiento de la función **mmu_mapear_página** consiste en :

- ▷ Obtener el offset del Page_directory (*directory*) shifteando a la derecha 22 bits la dirección virtual pasada por parámetro.
- ▷ Obtener el offset del Page_table (*table*) shifteando a la derecha 12 bits la dirección virtual pasada por parámetro luego de haberle hecho un *and* con 0x003FF000 así obtengo sólo los bits de interés.
- ▷ Si el índice *directory* del Page Directory tiene el bit presente = 0, entonces debemos setear la base obteniendo una nueva Page Table (con la función get_page_table), dándole permismos de escritura y nivel acorde a lo pasado por parámetro y seteando el bit de presente.
- ▷ Obtener el puntero a la Page Table (*pt*), accediendo al índice *directory* del Page Directory pasado por parámetro, lo shifteamos a la izquierda 12 bits así tenemos la dirección de la página donde se encuentra la Page Table.
- ▷ En el índice *table* de la Page Table apuntada por *pt* le asignamos la dirección física pasada por parámetro shifteada a la derecha 12 bits, asignándole permisos de lectura/escritura acorde y nivel a lo pasado por parámetro y seteando el bit de presente.
- ▷ Por último, ejecutamos *tlbflush* para que se invalide la cache de traducción de direcciones.

A veces, vamos a necesitar des-mapear una página para que, bajo el cr3 actual, no se tenga más permiso de acceso a la misma. Por este motivo, contamos con la función *mmu_unmapear_pagina*.

La función **mmu_unmapear_pagina** va a recibir como parámetro una dirección virtual y el cr3 actual. El comportamiento de esta función, si el bit de presente del Page Directory al que apunta al cr3 está seteado, va a consistir en el simple acceso al Page Directory apuntado por el cr3, accediendo al índice descripto en la dirección virtual pasada por parámetro. Luego dirigirse al Page Table apuntado por este, accediendo al índice que se encuentra en la dirección virtual y una vez ahí limpiar el bit de presente. En otro caso, no es necesario tomar ninguna acción.

5. Ejercicio 5

a) Entradas necesarias en la IDT.

Completamos las entradas necesarias en la IDT para asociar una rutina a la interrupción del reloj, otra a la interrupción de teclado y por último una a la interrupción de software 0x66. Es decir las posiciones 32, 33 y 102.

Las primeras dos van a ser de privilegio 0 y la última de privilegio 3, asignándoselo mediante sus atributos.

b) Rutina asociada a la interrupción de reloj.

La rutina asociada a la interrupción del reloj por cada tick llama a la función *próximo_reloj*. La misma se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla.

Su funcionamiento consiste en pushear los registros, llamar a *proximo_reloj* que se encarga de cambiar el caracter en pantalla y a proximo_indice el cual indica qué índice en la *GDT* es el próximo a ejecutar. Si debemos cambiar de *tss*, realiza el *jump far*, sino termina. En ambos casos, llama previamente a *fin_intr_pic1* y luego hace *popad* de los registros y vuelve con un *iret*.

c) Rutina asociada a la interrupción de teclado.

La rutina asociada a la interrupción de teclado fue programada en esta instancia de forma que si se presiona cualquiera de las teclas a utilizar en el juego, se presenta la misma en la esquina superior derecha de la pantalla.

El comportamiento de este handler consiste en testear qué tecla fue presionada (mediante *in **al**, 0x60*), llamar a la función handle_keyboard_interrumption que recibe como parámetro **eax**. Esta función se encarga de imprimir en pantalla el char que corresponda a la tecla presionada. Luego el handler llama a *fin_intr_pic1* y termina la ejecución un *iret*.

Luego, modificamos esta rutina y lo que ejecuta es:

- pushad
- Testea la tecla presionada, si es la *y* setea la variable que indica si está activo el debugger sino ejecuta *handle_keyboard_interrumption*.
- popad

El **debugger** se va a comportar acorde a lo explicado más adelante.

Si entra en el **handle_keyboard_interrumption**, lo que sucede es testear hacia donde es el movimiento y de que jugador para luego llamar a la función *movimiento* que mueva toda la paginación y lo mueva en pantalla.

d) Rutina asociada a la interrupción 0x66.

Escribimos la rutina asociada a la interrupción 0x66 para que modifique el valor de eax por 0x42, retornando con un *iret*.

6. Ejercicio 6

a) Entradas en la GDT usadas como descriptores de TSS.

Definimos cuatro entradas en la GDT que consideramos necesarias para ser usadas como descriptores de TSS: una para ser utilizada por la tarea inicial, una para la tarea Idle, otra para la tarea actual y una última para la tarea siguiente.

Cada una de ellas se encuentra en los siguientes índices:

```
#define GDT_INITIAL_TSS         0xD
#define GDT_CURRENT_TSS         0xE
#define GDT_NEXT_TSS            0xF
#define GDT_TSS_IDLE            0x10
```

b) Entrada de la TSS de la tarea Idle.

Completamos la entrada de la TSS de la tarea Idle con la información de la tarea Idle. La tarea Idle se encuentra en la dirección 0x00016000. La pila se aloja en la misma dirección que la pila del kernel y es mapeada con identity mapping. Esta tarea ocupa 1 pagina de 4KB y está mapeada con identity mapping. Además la misma comparte el mismo CR3 que el kernel.

Se le asigna de base 0xFFFFFFFF POR QUE???????. Al **eip** se le asigna el valor de 0x00016000, al **cr3** se le asigna el cr3 del kernel. Como la pila se setea en la posición 0x27000 se les adjudica a **ebp** y **esp** ese valor. Los registros **es**, **ds**, **ss** y **gs** se les da el valor de 0x40 mientras que al **cs** el de 0x50 Aca habria que explicar porque, no??. Los **eflags** se ponen en 0x202 así quedan habilitadas las interrupciones (es por eso?). Y por último se le asigna a **esp0** también 0x27000 y a **ss0** 0x40.

c) Completar una TSS libre con una tarea (Zombie).

Construir una función que complete una TSS libre con los datos correspondientes a una tarea (zombi). El código de las tareas se encuentra a partir de la dirección 0x00010000 ocupando una pagina de 4kb cada una según indica la figura 1. Para la dirección de la pila se debe utilizar el mismo espacio de la tarea, la misma crecerá desde la base de la tarea. Para el mapa de memoria se debe construir uno nuevo utilizando la función mmu inicializar dir zombi. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva pagina libre a tal fin.

d) Primera ejecución de la Tarea Idle.

El código necesario para saltar a la tarea Idle desde la ejecución de la tarea inicial se realiza con un salto cambiando las tss. Es llevado a cabo con el siguiente código?.

```
jmp 0x80:0
```

7. Ejercicio 7

a) Inicializar las estructuras de datos del scheduler.

En nuestro diseño contamos con las siguientes variables para llevar el control sobre el scheduler, qué jugador está jugando, cuál es el puntaje, cuántos zombies fueron lanzados y cuántos zombies hay activos:

- Puntaje de A y B, *inicializados en cero*.
- Posición actual de A y B en el eje de las *y*, *inicializados en 20 (mitad de la pantalla)*.
- Clase de zombie actual de A y B, *inicializados en MAGO*.
- Cantidad de zombies disponibles para lanzar de A y B, *inicializados en 20*.
- Cantidad de zombies activos de A y B, *inicializados en 0*.
- Variable que indica si el debugger está activo o no, *inicializado en FALSE*.
- Variable que indica si terminó el juego, *inicializado en FALSE*.
- Contador de cuanto tiempo hay de inactividad para terminar la ejecución, *inicializado en 0*.

b) Implementación próximo índice.

Implementamos la función **próximo índice** que devuelve el próximo índice en la GDT con la tss a ejecutar.

La función tiene el siguiente comportamiento:

- Si la CURRENT_TSS no está bussy, devuelve esta sino devuelve NEXT_TSS.
- Si el juego terminó devuelve la IDLE_TSS.
- Si la próxima tss a ejecutar es la misma que la actual, se devuelve 0 (así no se ejecuta el cambio de tarea que sería inválido).

c) Rutina de atención a la interrupción 0x66.

La rutina de atención a la **interrupción 0x66** se basa en pushear los registros, llamar a la función *game_move_current_zombi* y luego hace *pop* finalizando con un *jmp far* y un *iret*.

La función **game_move_current_zombi** consiste en:

- Asigna el valor 0 al contador de inactividad.
- Evalúa cuál es el movimiento a realizar (ADE, ATR, DER o IZQ) y luego se copia el codigo de la tarea en la próxima página. Se mapean las nuevas páginas y se adelanta al zombie en pantalla.
- Si el zombie murió se suma el punto correspondiente.

En la función que ejecuta el movimiento, para saber a qué jugador se corresponde se compara la dirección física correspondiente al centro con la que está mapeada "adelante"; si es menor es el Jugador B.

d) Rutina de atención a la interrupción del Clock.

La rutina de atención de Interrupción del clock consiste en:

- Se fija si está seteada la variable del debuggerOn pero no está habilitado el debugger (es decir debuggerOn2 == FALSE), si es así activa la pantalla de debugg. En cambio si debuggerOn == TRUE y debuggerOn2 == TRUE, salta al final.
- En caso de tener que activar el debugger, mediante el macro TAKE_SNAPSHOT se copian todos los valores de los registros y se apilan antes de llamar a la función que va a imprimir el debugger en pantalla.
- Si es un caso contrario a los considerados anteriormente, lo que se hace es un call a: *proximo_reloj*, *revisarTerminacion* y *proximo_indice*. Luego se hace el jmp far adecuado.
- En todos los casos, se llama previamente a la función fin_intr_pic1.

La función *proximo_reloj* se encarga de cambiar el caracter en pantalla que marca a los zombies activos (relojito). La función *revisarTerminacion* lo que hace es fijarse si terminó la ejecución del juego. Y por último, la función *proximo_indice* indica qué índice en la *GDT* es el próximo a ejecutar.

e) Rutina de atención a las excepciones del procesador.

El funcionamiento de esta rutina se basa en:

- Almacenar en memoria el estado de todos los registros y luego almacenarlos en la pila para ser pasados como parámetros.
- Llamar a la función desalojarTarea.
- Hacer un jmp far a la tarea Idle
- LLamar a la función print_exception.

La función **desalojarTarea** funciona del siguiente modo:

- Si la tarea NEXT_TSS está en Busy, esta es la que va a ser desalojada caso contrario va a ser la CURRENT_TSS.
- Compara con las tss guardadas del jugador A, para ver si le pertenece a él la tarea a desalojar. Si es así, la encuentra y le limpia el valor de en uso a cero, el clock de zombies de esta tarea fija en cero y se resetea su clock.
- Si no pertenecía al jugador A, se busca en el jugador y se hace lo mismo que lo explicado arriba.

La función **print_exception** es la explicada en el *Ejercicio 2*.

f) Mecanismo de Debugging.

Al momento de llamar a la interrupción del teclado con la tecla *y*, se chequea si ya estaba activado el debugger o no (esto es con la variable debuggerOn). Si no lo estaba, se almacena en memoria el contenido de toda la pantalla, de este modo se salva la pantalla previa a que se abra la ventana de Debugging. Además se le indica a la variable debuggerOn en TRUE y a la variable debuggerOn2 en FALSE, de este modo se asume que se activó el Modo de Debugg, pero todavía no se mostró en pantalla.

Si el debuggerOn hubiera estado en TRUE, lo que hace esta función es volver a copiar en el área del Mapa, la pantalla que había sido guardada previamente en memoria.

Cuando llega la interrupción del reloj, esta chequea ambas variables y cuando lo haga si la variable debuggerOn == TRUE y debuggerOn2 == FALSE su comportamiento va a ser llamar a la función *show_debugger*. En caso contrario, su comportamiento va a ser acorde a lo explicado anteriormente.

Contamos con un Macro que almacena en memoria todos los valores de registros al momento de la interrupción, de este modo la función que imprime en pantalla la ventana de Debugging (*show_debugger*) recibe como parámetro los mismos.

La función **show_debugger** va a ser la encargada de "pintar" la pantalla con la ventana de debugger, imprimiendo los valores de los registros que fueron pasados por parámetro.